알고리즘 설계와 분석 프로그래밍 숙제 1

2반

20141494

강동욱

1. 수행 환경

OS: Window 10 Education

CPU: AMD Ryzen 5 1600 3.70GHz

RAM: 16.00GB

Compiler: Visual Studio Community 17 (v141)

2. 실험 개요

이번 숙제에서는 수업 시간에 배운 선택 (selection)과 정렬 (sorting) 방법을 구현하여 이론적으로 배운 내 용과 실제로 구현한 결과를 비교하여 본다. 특히 교과서적인 quick sort 방법의 속도 향상을 위하여 속도 최적화 기법을 적용한 후 분석하여봄으로써 고급 소프트웨어 개발능력을 배양토록 한다.

3.구현 요구사항

i. Insertion sort: 전형적인 insertion sort 방법을 구현함.

ii. Heap sort: 전형적인 heap sort 방법을 구현함.

iii. qsort: 이 함수는 직접 입력 데이터를 정렬하지 않고 Standard C library가 제공하는 qsort() 함수를 호출하여 원하는 정렬 계산을 수행함.

iv. Quick sort: 자신만의 효과적인 pivot strategy를 사용하여 pivot 원소를 선택하는 재귀적인 방식 의 전형적인 quick sort 방법을 구현함.

v. Optimized quick sort: 바로 위에서 구현한 방법을 확장하여 최적화된 quick sort 방법을 구현하 라.

4. 실험 결과.

(a) Entirely random: 배열 원소의 key 값으로 적절한 범위의 random number를 n개 생성하여 만든 데이터.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Data size | Insertion sort | Heap sort | qsort | Quick sort |
| 1024 | 0.889 | 0.922 | 0.075 | 0.277 |
| 4096 | 13.412 | 4.705 | 0.291 | 1.191 |
| 16384 | 215.085 | 19.405 | 1.201 | 5.246 |
| 65536 | 3530.778 | 88.697 | 4.560 | 23.112 |
| 262144 | 58268.313 | 428.568 | 18.662 | 101.507 |
| 1048576 |  | 2073.204 | 72.418 | 443.152 |

(시간 단위:milliseconds)

(b) Descending: 배열 원소의 key 값이 n−1, n−2, n−3, · · · , 1, 0와 같이 정반대로 정렬되어 있는 배열 데이터.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Data size | Insertion sort | Heap sort | qsort | Quick sort |
| 1024 | 1.686 | 0.956 | 0.073 | 0.101 |
| 4096 | 26.823 | 4.354 | 0.289 | 0.418 |
| 16384 | 433.633 | 19.807 | 1.142 | 1.755 |
| 65536 | 7146.660 | 88.363 | 5.186 | 7.586 |
| 262144 | 128287.875 | 412.016 | 20.575 | 31.679 |
| 1048576 |  | 1767.943 | 78.348 | 136.127 |

(시간 단위:milliseconds)

(c) Few swaps: 배열 원소의 key 값이 0, 1, 2, · · · , n − 1과 같이 정렬된 상태에서 int(√ n)개의 random number pair (i, j)를 생성하여 i번째 원소와 j번째 원소를 서로 교환함 (이때 |i − j| < √ n의 조건을 만족하도록 할 것).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Data size | Insertion sort | Heap sort | qsort | Quick sort |
| 1024 | 0.056 | 0.965 | 0.076 | 0.086 |
| 4096 | 0.155 | 4.596 | 0.284 | 0.345 |
| 16384 | 0.651 | 21.535 | 1.132 | 1.430 |
| 65536 | 3.391 | 97.695 | 4.575 | 5.919 |
| 262144 | 9.982 | 419.245 | 18.094 | 24.951 |
| 1048576 |  | 1890.671 | 73.478 | 106.743 |

(시간 단위:milliseconds)

2. 과연 수행 시간과 이 방법들의 이론적인 시간 복잡도간에 깊은 연관성을 발견할 수 있는가?

Insertion sort는 Entirely random과 Descending의 경우 Data size의 증가에 따른 수행시간의 변화가 제곱으로 비례하면서 의 시간복잡도와 깊은 연관성을 보이지만, 데이터가 거의 정렬되어있는 Few swaps의 경우에는 속도가 거의 에 가깝게 나타났다.

Heap sort의 경우 이론상 시간 복잡도는 으로 모든 경우에서 n이 4배 증가 할 때 수행시간은 N이 2의 10제곱일 경우 4.8배, 12제곱일 경우 4.7배, 16제곱일 경우 4.5배 증가한다. 모든 경우에서 깊은 연관성을 보임을 확인 할 수 있다.

Qsort와 Quick sort의 경우 Heap sort와 이론상 시간복잡도는 으로 같지만, 거의 O(N)과 같은 Data size의 증가에 따른 수행시간의 변화를 보이는데, 지금의 수행시간 T(n)은 과 같이 나타낼 수 있는데 함수호출 과정의 횟수인 에 호출시간 상수를 곱한 가 보다 크기 때문에 발생했다고 볼 수 있다.

3.이때 서로 다른 종류의 데이터에 대하여 수행 시간에 차이가 있는가? 있다면 그 이유는 무엇으로 추정되는가?

Insertion sort의 경우 정렬된 리스트에 값을 추가 할 때 sort과정의 대부분을 차지하는 insertion 과정이 매우 적게 일어나기 때문에, 가장 정렬된 Few swaps 데이터에서는 매우 빠른 속도를 보여주지만, 정렬 상태와 반대로 정렬된 Descending 데이터에서 최악의 수행시간을 보여준다.

Heap sort, qsort의 경우 다른 정렬법에 비해 데이터의 종류에 따라 시간차이가 적었다.

Quick sort의 경우 Descending, Few swaps의 경우의 수행시간과 Entirely random의 경우 수행시간이 많은 차이가 발생했는데, Quick sort의 경우 pivot을 행렬의 제일 앞 (인덱스 left), 가운데 (인덱스 (left+right)/2), 제일 뒤 (인덱스 right)에 있 는 세 원소들 중앙값을 가지는 원소를 pivot 원소로 사용하는데, Descending, Few swaps의 경우 정렬되어 있기 때문에 pivot이 대부분 중간 값으로 선정되게 되어 효율적이지만, Entirely random 의 경우 정렬되지 않았기 때문에 상대적으로 중간값에 가깝지 않은 pivot값이 선정되어 이런 결과가 나타난다.

4. 과연 quick sort 방법은 그 이름이 의미하는 바와 같이 정말 빠른가?

특정한 상황에서는 가장 빠르지는 않았다. 특히 거의 정렬된 행렬에서는 insertion sort에 비해 상당히 느리게 나타났다.

나머지 경우에는 다른 sort에 비해 큰 차이로 빠른 것을 확인 할 수 있다.

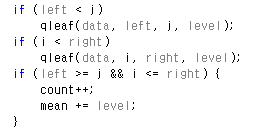
5. 과연 어떤 부류의 데이터에 대해 insertion sort 방법의 수행 시간이 heap sort 또는 quick sort 방법 에 비해 빠르거나 그리 느리지 않은가? 과연 insertion sort 방법이 거의 선형적인 시간에 작동하는 경우를 발견하였는가?

Few swaps 처럼 거의 정렬되어있고, 데이터의 일부만 정렬되지 않았을 경우에 거의 선형적인 시간에 작동하며, 다른 sort보다 훨씬 빠르게 수행되었다.

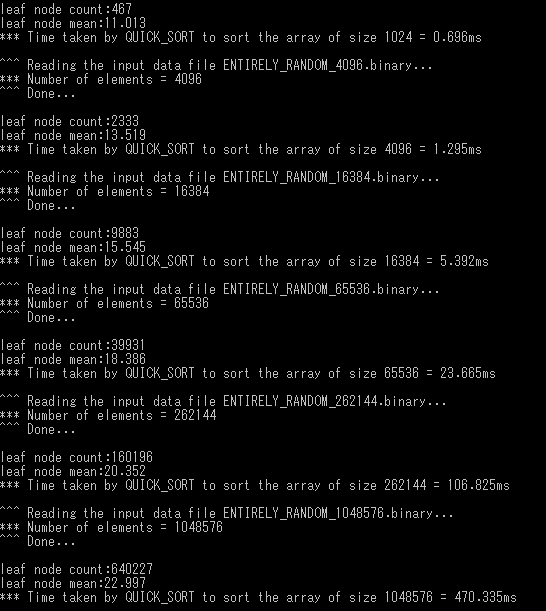
6. n 값이 작을 경우 굳이 quick sort 방법을 사용할 필요 없이 insertion sort 방법을 사용해도 크게 문제 가 되지 않는다. 과연 이는 어느 정도 범위의 n 값까지일지 Entirely random 데이터에 대한 실험을 통하여 자신의 값을 제시하라.

Qsort의 경우 대략 n이 64까지는 insertion sort가 더 빠르거나 같은 속도를 보였고, Quick sort의 경우 대략 n이 128까지 insertion sort가 더 빠르거나 같은 속도를 보였다.

7. 입력 데이터의 크기 n에 quick sort 방법을 수행할 때 재귀적으로 생성되는 트리의 깊이가 어느 정도 되는지 각 리프 노드의 레벨에 대한 최소/최대/평균을 적절히 크기가 큰 Entirely random 데이터에 대한 실험을 통하여 자신의 값을 제시하라. 평균적으로 최악의 경우에 해당하는 O(n)에 가까운지 아니면 수업 시간에 이론적으로 증명한 바와 같이 O(n log n)에 가까운지 실험적으로 분석할 것.



위와 같은 코드로 재귀의 level을 기록하면서 리프노드일 경우 리프노드의 개수를 증가시키고, 그 level의 합을 기록하여 나중에 평균을 구한 결과,



Leaf node mean 으로 표시된 깊이의 평균은 n이 일 떄보다 대략 2배 큰 것을 확인할 수 있으며, 각 평균값과 n과의 관계를 분석하면, 리프 노드의 레벨은 O(logn)에 가까운 모습을 보이는 것으로 나타난다.

8. 각 방법을 최대한 효율적으로 구현하기 위하여 어떤 노력을 기울였는가?

Insertion sort의 경우 데이터를 하나씩 이동시키지 않고, memcpy를 통하여 바이너리 데이터를 이동시킬 사이즈\* ELEMENT의 크기만큼 한번에 복사하여 시간을 줄였다.

Quick sort의 경우 pivot select에서 세 원소 중 두 번째로 큰 원소를 뽑는 과정을 inline 함수로 처리하고, 조건문 중첩을 사용하지 않고 비트 연산자를 사용하여 빠르게 세 원소중 두번째로 큰 원소를 구해냈다.

9. 기타 이번 숙제를 하면서 적용한 창의적인 방법이나 경험적으로 알게 된 중요 사항을 기술하라.

세 원소 중 두 번째로 큰 원소를 뽑는 방법을 if문이나 삼항 연산자로 구하는 것이 너무 길어지기에 껄끄러워서 찾아보니 min(A,B) XOR min(A,C) XOR min(B,C)와 같이 비트연산자를 통하여 가장 작은 원소를 XOR로 묶어 0으로 만들고 0과 두 번째로 작은 원소를 XOR 연산하여 구하는 방법을 보고 감탄하게 되었다. 이 방법이라면 조건문보다 훨씬 빠르게 구할 수 있을 것이다.

5. (속도 최적화 컨테스트) 조교는 여러분이 구현한 함수 QUICKSORT OPT() 함수에 대하여 적절히 크기가 큰 Entirely random 데이터를 사용하여 수행 시간을 측정한 후 속도 순으로 상위 10% – 20% 정도 (정확한 숫자는 상대적인 상황에 따라 추후 결정)의 학생에게 100점 만점 대비 최대 20점에 해당하는 추가 점수를 부여할 수 있다.

1. 데이터가 size가 50이하에선 insertion sort를 시행하도록 바꿈

2. 꼬리재귀 최적화를 시행하였으나 시행시간에는 변화 없어 되돌림

3. pivot selection에서 selection\_rec사용해봤으나 느려짐

4. pivot selection을 랜덤으로 지정해봤으나 평균적으로 느려짐

5. void함수 바꿔서 꼬리재귀를 시행해 봤으나 변화없음

6. 프로젝트설정을 런타임에러를 체크하지 않고 속도 최적화로 바꾸었으나 프로젝트단위로 측정하지 않는 듯함

7. 그냥 std::sort를 사용하는 건 의도와 맞지 않는듯함